Linked-List 1

Data Structures

Data structures are just a way to store and organize our data so that it can be used and retrieved as per our requirements. Using these can affect the efficiency of our algorithms to a greater extent. There are many data structures that we will be going through throughout the course, linked-lists are a part of them.

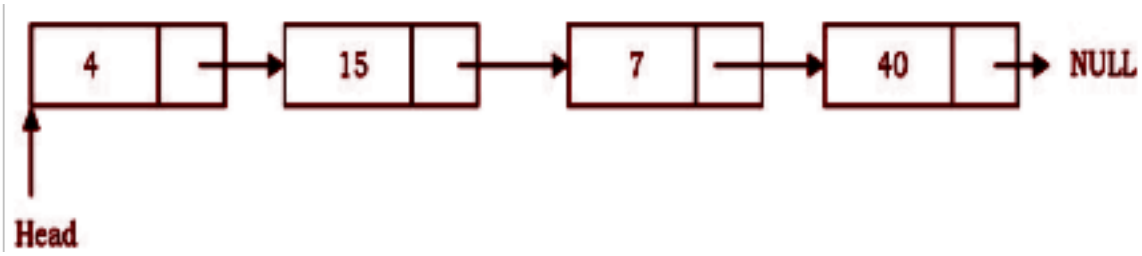
Introduction

A **Linked List** is a data structure used for storing collections of data. A linked list has the following properties:

- Successive elements are connected by pointers.
- Can grow or shrink in size during the execution of a program.
- Can be made just as long as required (until systems memory exhausts).
- Does not waste memory space (but takes some extra memory for pointers). It allocates memory as the list grows.

Basic Properties:

- Each element or node of a list is comprising of two items:
    - Data
    - Pointer(reference) to the next node.
- In a Linked List, the elements are not stored at contiguous memory locations.
- The first node of a linked list is known as **Head**.
- The last node of a linked list is known as **Tail.**
- The last node has a reference to null.



Types of A Linked List

- **Singly-Linked List:** Generally "linked list" means a singly linked list. Each node contains only one link which points to the subsequent node in the list.

- **Doubly-Linked List:** It's a two-way linked list as each node points not only to the next pointer but also to the previous pointer.

- **Circular-Linked List:** There is no tail node i.e., the next field is never **null** and the next field for the last node points to the head node.

- **Circular Doubly-Linked List:** Combination of both Doubly linked list and circular linked list.

Node Class (Singly Linked List)

```
// Node class
class Node{
    int data;
    Node next;
    // Function to initialize the node object
    Node(int data){
        this.data = data;  // Data that the node contains
        this.next = null; // Next node that this node points to
    }
}
```

Note: The first node in the linked list is known as **Head** pointer and the last node is referenced as **Tail** pointer. We must never lose the address of the head pointer as it references the starting address of the linked list and if lost, would lead to loss of the list.

Traversing the Linked List

Let us assume that the head points to the first node of the list. To traverse the list we do the following:

- Follow the pointers.
- Display the contents of the nodes (or count) as they are traversed.
- Stop when the next pointer points to **null**.

Printing the Linked List

To print the linked list, we will start traversing the list from the beginning of the list(head) until we reach the **null** pointer which will always be the tail pointer. Let us add a new function `printList()` to our **LinkedList** class.

```
// This function prints contents of linked list starting from head
public static void printList(Node headNode){
    Node temp = headNode; //Start from the head of the list
    while (temp != null){ //Till we reach the last node
        System.out.print(temp.data + " ");
        temp = temp.next; //Update temp to point to the next Node
    }
}
```

Insertion of A Node in a Singly Linked List

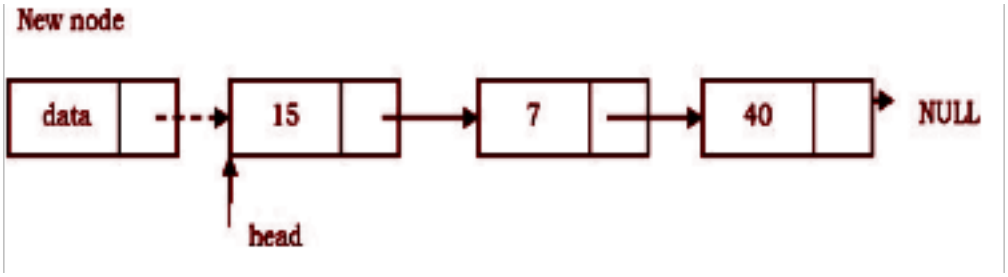
There are 3 possible cases:

- Inserting a new node before the head (at the beginning).
- Inserting a new node after the tail (at the end of the list).
- Inserting a new node in the middle of the list (random location).

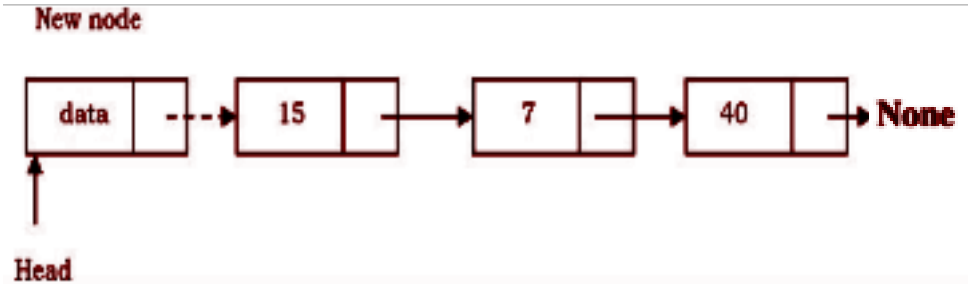
Case 1: Insert node at the beginning:

In this case, a new node is inserted before the current head node. Only one next pointer needs to be modified (new node's next pointer) and it can be done in two steps:

• Create a new node. Update the **next** pointer of the new node, to point to the current head.



• Update **head** pointer to point to the new node.



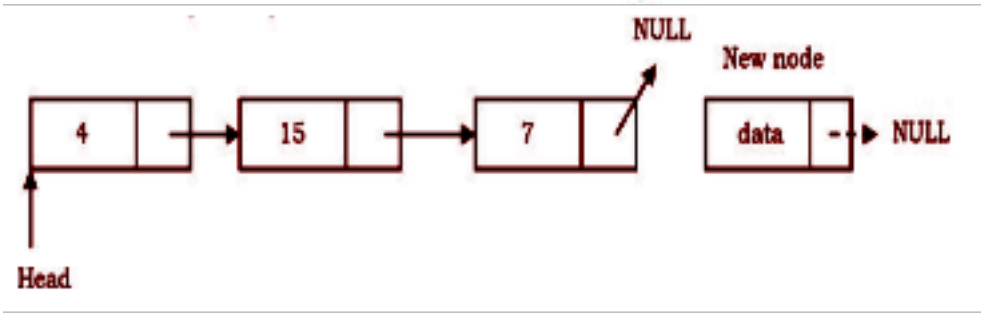
Java Code:

```
public static Node insertAtStart(Node head, int data){
    Node newNode = new Node(data); //Create a new node
    newNode.next = head; //Set next node of new node to current head
    head = newNode; //Update the head pointer to the new node
    return head;
}
```

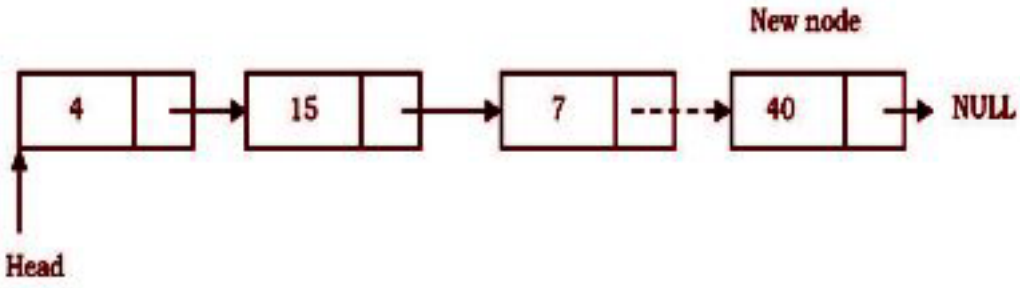
Case 2: Insert node at the ending:

In this case, we need to modify two next pointers (last nodes next pointer and new nodes next pointer).

• New node's **next** pointer points to **null**.



• Last node's **next** pointer points to the new node.



Java Code:

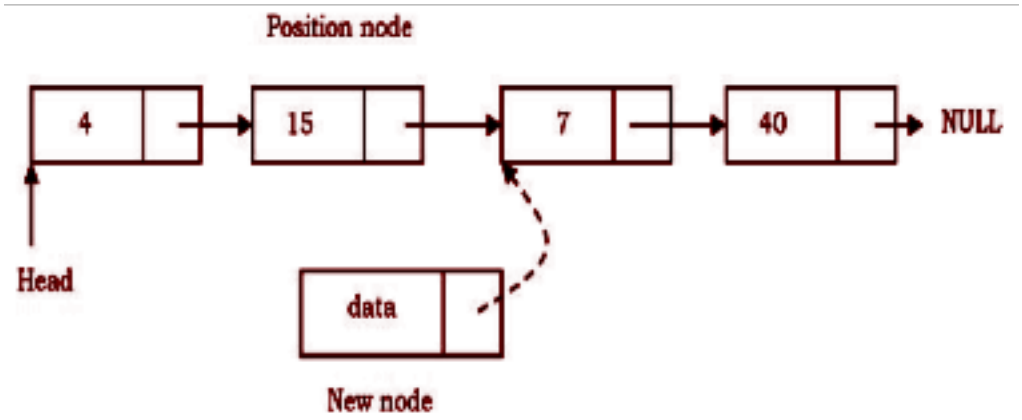
```
public static Node insertAtEnd(Node head, int data){
        Node newNode = new Node(data); //Create a new node
        if (head == null){ //Incase of empty LL
            head = newNode;
            return;
        }
        Node n = head;
        while (n.next != null) //If not empty traverse till last node
            n = n.next;
        n.next = newNode; //Set next = new node
        return head;
}
```

Case 3: Insert node anywhere in the middle: (At any specified Index)

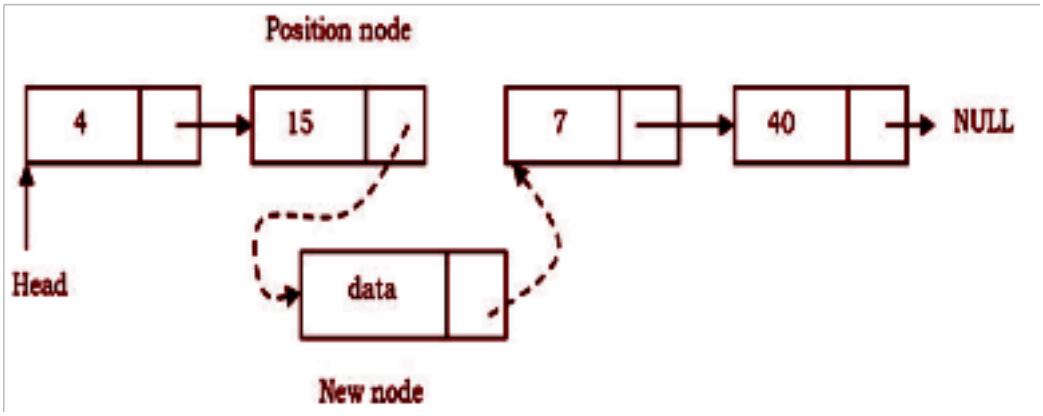
Let us assume that we are given a position where we want to insert the new node. In this case, also, we need to modify two next pointers.

• If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node.

• For simplicity let us assume that the second node is called the **position node**. The new node points to the next node of the position where we want to add this node.



• Position node's **next** pointer now points to the new node.



Java Code:

```
public static Node insertAtIndex (int head, int index, int data){
        if (index == 1)  // Insert at beginning
            insertAtStart(head, data);
        int i = 1;
        Node n = head;
        while (i < index-1 && n != null){
            n = n.next;
            i = i+1;
        }
        if (n == null)
            print("Index out of bound");
        else{
            Node newNode = new Node(data);
            newNode.next = n.next;
            n.next = newNode;
        }
}
```

Deletion of A Node in a Singly Linked List

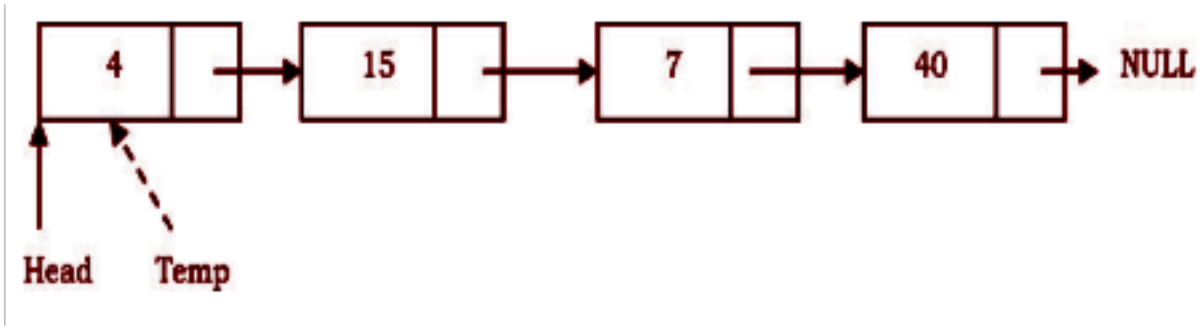
Similar to insertion, here we also have three cases.

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node.

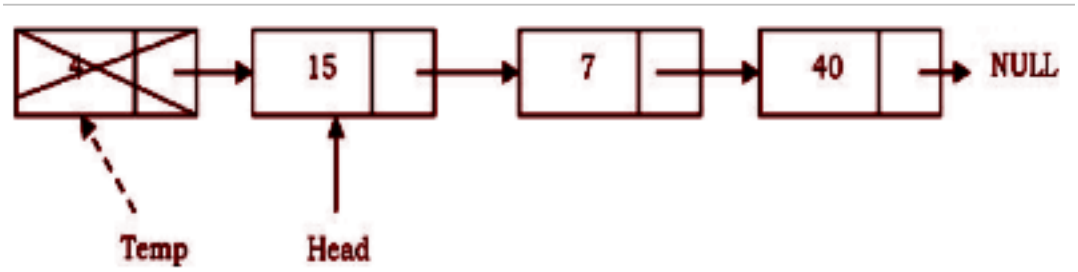
Deleting the First Node in Singly Linked List

It can be done in two steps:

• Create a temporary node which will point to the same node as that of the head.



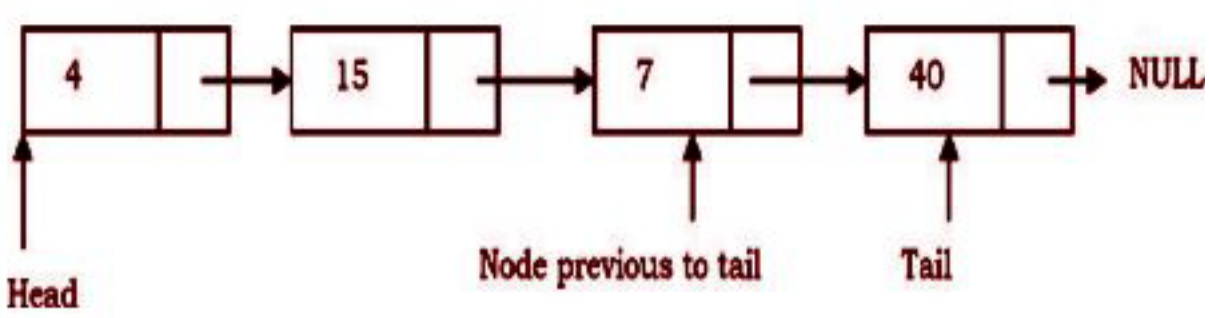
• Now, move the head nodes pointer to the next node and dispose of the temporary node.



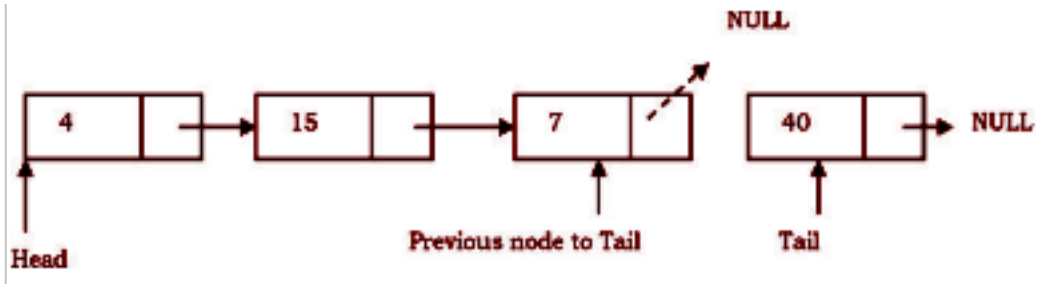
Deleting the Last Node in Singly Linked List

In this case, the last node is removed from the list. This operation is a bit trickier than removing the first node because the algorithm should find a node, which is previous to the tail. It can be done in three steps:

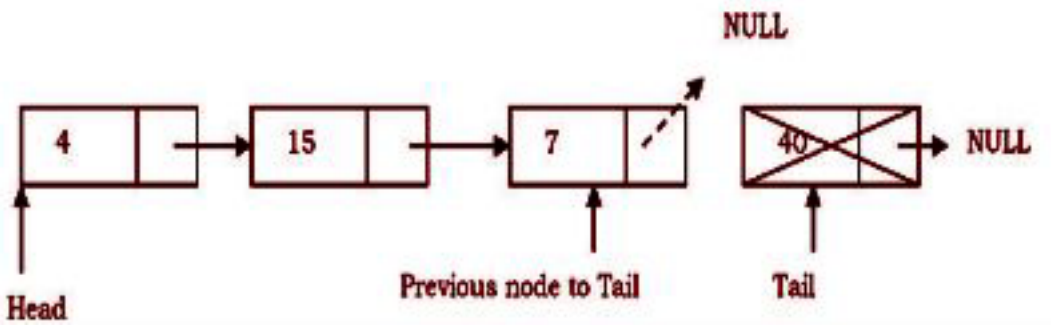
• Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail node and the other pointing to the node before the tail node.



• Update the previous node's next pointer with **null**.



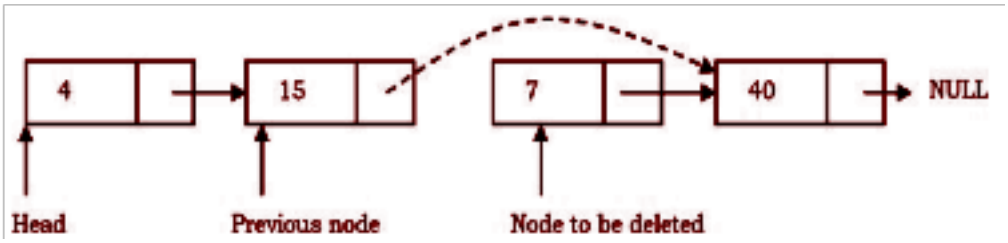
• Dispose of the tail node.



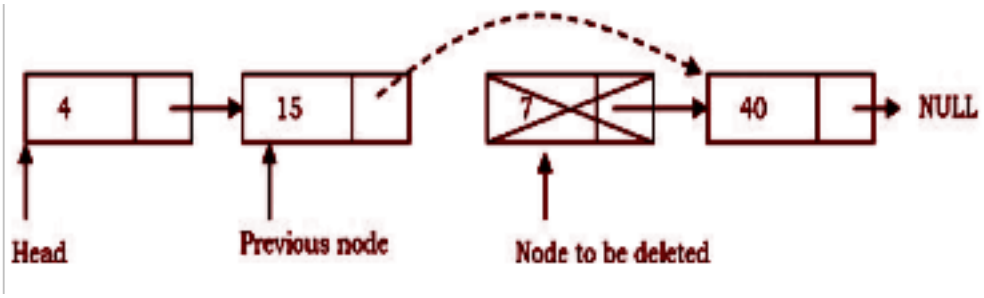
Deleting an Intermediate Node in Singly Linked List

In this case, the node to be removed is always located between two nodes. Head and tail links are not updated in this case. Such removal can be done in two steps:

• Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.



• Dispose of the current node to be deleted.



Insert node recursively

Follow the steps below and try to implement it yourselves:

- If **Head** is **null** and **position** is not 0. Then exit it.
- If **Head** is **null** and **position** is 0. Then insert a new Node to the **Head** and exit it.
- If **Head** is not **null** and **position** is 0. Then the **Head** reference set to the new Node. Finally, a new Node set to the Head and exit it.
- If not, iterate until finding the Nth position or **end**.

For the code, refer to the Solution section of the problem.

Delete node recursively

Follow the steps below and try to implement it yourselves:

- If the node to be deleted is the **root**, simply delete it.
- To delete a middle node, we must have a pointer to the node previous to the node to be deleted. So if the position is not zero, we run a loop position-1 times and get a pointer to the previous node.
- Now, simply point the previous node's next to the current node's next and delete the current node.

For the code, refer to the Solution section of the problem.